Securing Time in Untrusted Operating Systems with TimeSeal

Fatima M. Anwar
UMass Amherst
fanwar@umass.edu

Luis Garcia
UCLA
garcialuis@ucla.edu

Xi Han
UCLA
xihan94@ucla.edu

Mani Srivastava
UCLA
mbs@ucla.edu

***Abstract*—An accurate sense of elapsed time is essential for the safe and correct operation of hardware, software, and networked systems. Unfortunately, an adversary can manipulate the system's time and violate causality, consistency, and scheduling properties of underlying applications. Although cryptographic techniques are used to secure data, they cannot ensure time security as securing a time source is much more challenging, given that the result of inquiring time must be delivered in a *timely* fashion.**

In this paper, we first describe general attack vectors that can compromise a system's sense of time. To counter these attacks, we propose a secure time architecture, TIMESEAL that leverages a Trusted Execution Environment (TEE) to secure time-based primitives. While CPU security features of TEEs secure code and data in protected memory, we show that time sources available in TEE are still prone to OS attacks. TIMESEAL puts forward a *high-resolution* time source that protects against the OS *delay* and *scheduling* attacks. Our TIMESEAL prototype is based on Intel SGX and provides sub-millisecond (msec) resolution as compared to 1-second resolution of SGX trusted time. It also securely bounds the relative time accuracy to msec under OS attacks.

In essence, TIMESEAL provides the capability of trusted timestamping and trusted scheduling to critical applications in the presence of a strong adversary. It delivers all temporal use cases pertinent to secure sensing, computing, and actuating in networked systems.

***Index Terms*—trusted clock, shielded execution, delay attack, scheduling attack, Intel SGX**

I. INTRODUCTION

Emerging temporal use cases in the Internet of Things (IoT) applications have a critical dependence on high precision and accurate *relative* time. For instance, distance and speed calculations rely on precise round trip times [1] [2], schedulers build upon elapsed and remaining times [3] [4], network telemetry depends on residency delays [5], code profiling requires execution times [6], and data sampling needs timestamps [7]. Attacking a system's sense of time has ramifications such as location theft, network outages, higher delays, and data inconsistencies. Furthermore, critical functionalities for securing applications in shielded execution environments such as Haven [8], SCONE [9], Panoply [10], and Graphene-SGX [11] have no access to secure time. Instead, they rely on untrusted operating system (OS) time. A compromised OS may lie about the time or signal early timeouts, and although traditional cryptographic techniques, trusted execution technologies, and network security mechanisms may guarantee data security; they do not cater to *time security*.

In an attempt to provide this security of time, researchers have tried to implement secure clocks for shielded execution in Trusted Execution Environments (TEE) [12]. A well known approach fTPM [13] tries implementing a secure clock on Arm TrustZone, but it relies on untrusted OS acknowledgements for clock writes. On the other hand, Déjá Vu [6] leverages hardware transactional memory to provide a high resolution clock for Intel Software Guard Extension (SGX), but it is susceptible to frequent aborts and delay attacks. While Aurora [14] leverages hardware support of System Management RAM (SMRAM) to provide an absolute clock for SGX, it still relies on untrusted timers and kernel devices. All of these clocks are prone to attacks in the presence of a compromised OS.

To reason about the generalized security of a clock, we argue that it is essential to secure all layers in a time stack. Providing this security of a clock gives rise to three main challenges: first, find a trusted timer that cannot be modified by a privileged adversary [15], second, provide a secure path to read the trusted timer in a timely manner [16], third, protect timekeeping software from adversarial attacks [6]. In this paper, we propose TIMESEAL, a secure time architecture designed to tackle these challenges.

To address the first challenge, we compare the timing capabilities of different Trusted Execution Environments (TEE) [12]. Based on our analysis, TIMESEAL leverages hardware-based protection of Intel SGX that gives

2576-3172/19/$31.00 ©2019 IEEE
DOI 10.1109/RTSS46320.2019.00018

access to a trusted timer. A privileged adversary in a compromised OS cannot write to the SGX trusted timer.

Concerning the second challenge, the SGX community has confirmed that the path to SGX trusted timer is not secure [17]. The SGX platform service transfers trusted time packets over a secure session via OS interprocess communication (IPC) [18]. While these packets are encrypted and integrity protected and a compromised OS cannot change the packet contents, the OS can still delay these packets and violate their timely arrival, consequently relaying the wrong elapsed time. We address this challenge of securing trusted timer access by mitigating the effects of this *delay attack* on trusted time values.

Unfortunately, SGX time – which increments once per second – is insufficient to protect against delay attacks. TIMESEAL builds upon coarse-grained SGX trusted timer and uses counting threads to increase its resolution while adopting compensation mechanisms to mitigate accumulative errors and maintain monotonicity. Our high resolution and monotonic SGX clock is capable of measuring intervals as small as 0.1msec and timestamp events that are apart by $\geq$ 0.1msec. TIMESEAL not only serves applications with high precision requirements [15] [19] but also utilizes this improved precision to detect and mitigate delay attacks.

The third and final challenge requires us to defend against attacks on timekeeping software, where a compromised OS may downgrade the time resolution to seconds by scheduling out the associated timekeeping threads. TIMESEAL overcomes these *scheduling attacks* and adopts multithreaded counting policies to induce uncertainty in malicious scheduling and reduce the efficacy of this attack on resolution degradation. As a result, the system maintains msec-level resolution that is also substantial to detect and compensate for delay attacks. Indeed there are applications in need of μsec-level precision while TIMESEAL achieves msec-level accuracy because of the fundamental hardware limitations of large timer access latency in commodity platforms.

Securing time in the presence of a compromised OS is a big challenge. This paper presents a thorough design exercise of securing time by highlighting the tradeoffs among performance, security, and resource consumption. Indeed both hardware and software-based design is needed; however, for an extensible solution, we provide a software-only approach leveraging existing hardware features from shielded execution to provide secure time. Though we leverage SGX for constructing a secure clock, TIMESEAL is not restricted to one architecture, and its principles are general enough to be applied to different hardware architectures.

Our contributions are summarized as follows,

- We provide a complete guideline for time security by enumerating challenges in securing a clock (§IV-A).
- We identify and verify that the path to reading SGX time is not secure by implementing a delay attack in OS and disrupting the timely arrival of SGX time (§IV-A).
- We present TIMESEAL, a secure time architecture that addresses the aforementioned challenges.
- We provide a high resolution SGX time (§V) and use the improved resolution to mitigate delay attacks (§V-B).
- We devise policies that reduce the effect of malicious scheduling on time error (§V-A).
- We prototype TIMESEAL on Intel SGX and evaluate the complete secure time architecture (§VI).
- We provide recommendations to hardware vendors that are critical to improving the accuracy of a secure clock (§VII).

This paper is organized as follows. We discuss the related work and preliminary information necessary to understand secure time in §II and §III, respectively. We present the design challenges, threat model and overview of TIMESEAL in §IV. We detail how TIMESEAL achieves a high resolution secure clock as well as how it overcomes different classes of attacks in §V. The implementation and evaluation of TIMESEAL are presented in §VI. We provide a discussion of the solution in §VII and conclude in §VIII.

II. RELATED WORK

Researchers have attempted to implement trusted clocks [20] [21] to enforce time-based policies [22]. For example, Chen et al. [23] provides coarse-grained secure clock on a trusted cloud with unrealistic assumptions that the network link has a bounded delay and a trusted counter is present in the cloud. Also, Raj et al. [13] face numerous challenges in implementing a secure clock on ARM TrustZone. Though a peripheral such as a timer can be mapped to the secure world, a peripheral's controller can still be programmed by the normal world [13]. The absence of a secure timer forces them to rely on a compromised OS to acknowledge clock writes and make the clock persistent.

TrustedClock [24] for SGX and Aurora [14] tries to provide a high resolution and absolute secure clock for SGX enclaves. Their absolute clock leverages System Management RAM (SMRAM) for timekeeping, and relies on a kernel daemon to trigger system management interrupt (SMI) for a clock read request. A malicious OS can make time fuzzy by arbitrarily delaying these requests. Also, SMI handler reads legacy timers on Intel Architecture that can be written to by the OS in a consistent manner to avoid detection. Thus relying on hardware timers that OS can manipulate and reading from kernel devices that can be delayed makes their system not secure.

Much closer to our work is an entire body of research that uses high resolution timers either to launch side channel attacks [16] [25] or to detect side channel attacks [6] in shielded execution. The absence of high resolution timers inside shielded execution environments, particularly Intel SGX, motivated researchers to find alternative solutions. Malware Guard Extension [16] emulates a short-lived precise timer inside an enclave to perform a Prime+Probe cache side-channel attack against co-located enclaves. Déjá Vu [6] identifies page faults by measuring execution time of the victim code in SGX with a reference-clock that is incremented within a Hardware Transactional Memory [26] to detect OS interruptions. Both solutions are affected by high frequency aborts and prone to OS delay attacks.

Because of the absence of a secure notion of time, most systems rely on untrusted time. For example, fine-grained cycle-level measurements are made inside enclaves via reasonably fast Network Interface Card (NIC) clock for line rate processing in middleboxes such as ShieldBox [15] and Slick [19]. However, as noted by the authors, the NIC clock is not secure against OS attacks. They argue that there is no precise trusted time source for shielded execution and it remains an open problem [15] [16]. Other secure systems such as SCONE [9], Haven [8], and Panoply [10] also rely on untrusted time via OS system calls.

In short, secure clocks are non-existent, and current secure systems have to rely either on untrusted clocks or coarse-grained clocks, leaving out many applications in need of precise trusted time.

III. Background

This section covers concepts that influence our design choices for TimeSeal. We start by explaining a traditional time stack in all systems, possible attacks on the stack, and timing capabilities of TEE.

Attacks on the Time Stack: Every system maintains a time stack. Discrete components that make up a time stack are hardware timers and timekeeping software. A hardware counter/timer counts the number of cycles of a periodic signal obtained from an oscillator. Timekeeping software maintains time by converting cycle counts into human understandable time.

A hardware timer in a time stack is not considered secure if a malicious software is able to write to its registers. In Intel architecture, RDTSC/RDTSCP results are not immune to influences by privileged software, e.g., the Time Stamp Counter (TSC) can be written to by the OS [27][28]. Similarly, other timers such as the High Precision Event Timer (HPET) can be controlled by the OS. Basically, the design principle of the OS dictates that its high privilege allows it to write to all registers. Hence, a diverse set of timers such as TSC, APIC, HPET, PIT present in Intel architectures are controlled by the OS. The diversity of these timers cannot be leveraged to detect misbehaving OS as the OS may remain undetected by consistently lying for all timers.

Virtualizing timer via trusted hypervisor also does not protect against timer modifications. First, OS can consistently alter timer to avoid detection. Second, no hypervisor can detect malicious time delays. Furthermore, OS is responsible for timekeeping, and it can lie about time, signal timeouts early or late, delay time transfer to applications, or gradually change the notion of time for applications to deceive them. Hence, a privileged software is capable of adding discontinuities in time. To protect these timers, it is essential to restrict timer writes only to trusted entities.

Trusted Time in SGX: Intel released Software Guard Extensions (SGX) which are a set of CPU instructions to secure code and data in protected memory. SGX's reserved & isolated memory regions called enclaves provide straightforward mechanism of SGX virtualization. Hence, SGX technology has been seamlessly adopted in the server market and paving its way towards mobile phones.

Intel SGX supports trusted time service by leveraging the security capabilities of Intel Converged Security and Management Engine (CSME) [18]. The CSME consists of an embedded hardware engine that runs its own firmware. This firmware allows the host to load and execute Java applets in the CSME at runtime through a dynamic application loader. SGX hosts different architectural enclaves that provide key services to application enclaves. Platform Service Enclave (PSE) is an architectural enclave that provides trusted time and monotonic counter service. Architectural Enclave Service Manager (AESM) is a background service that hosts the PSE and automatically loads and starts the Platform Services DAL Applet (PSDA) inside CSME to securely expose the CSME battery backed Protected Real-Time Clock

(PRTC) timer. As CSME is backed up with a battery, it's not affected by CPU power management states, i.e., it always has power to keep the PRTC running. PSDA and PSE communicate through a Management Engine Interface (MEI) driver that supports data exchange through a secure, memory-mapped mechanism not accessible by OS.

Intel SGX gets its notion of time from the PRTC timer in CSME. The OS can neither access nor manipulate the PRTC and, hence, SGX provides access to a trusted timer. This trusted time is managed by a PSE that reads the PRTC and transforms it into *SGX time* by appending an epoch to it. An application enclave first establishes a secure session with PSE, then invokes the *sgx_get_trusted_time()* API to get SGX time. This time is in seconds and too coarse-grained to be useful for measuring short durations within a single enclave.

An application accesses SGX time through the encrypted and integrity protected PSE messages passing through the OS layer. These messages can be captured or replayed by the OS. However, a PSE message includes a sequence number that helps reset a session if replay attacks are detected.

IV. TIMESEAL DESIGN

To the best of our knowledge, this is the first paper that presents a comprehensive list of attacks on all components in a time stack for various architectures. It establishes that applications in shielded execution are also affected by timing attacks. We categorize our list of attacks and their mitigation such that they cover all components in a time stack. To provide a secure time architecture, the addressed challenges are namely the availability of a trusted timer, secure access to that timer, and secure clock software. In this section, we present design overview of TIMESEAL overcoming these challenges under a generalized threat model.

A. Challenges

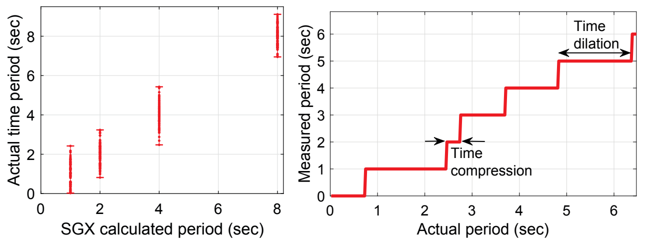
The choice of a trusted timer, one that cannot be modified by a privileged adversary, is critical for TIMESEAL's design. To make this choice, we compare the timing capabilities of SGX and TPM, the only two trusted clocks provided by different TEE.

`[Challenge 1]` A Trusted Timer - SGX Trusted Time versus TPM Secure Clock: There are a number of limitations in providing secure time over Intel SGX. First, peripheral registers such as hardware timer's registers cannot be mapped into protected enclave memory to protect the timer from OS manipulation. Only the Process Reserve Memory (PRM), a special DRAM for secure enclaves in SGX, can be mapped into the virtual address of an enclave page – or in SGX's terminology – Enclave Page Caches (EPC). Second, SGX trusted timer has coarse-grained, one-sec resolution [18] that can only be used for enforcing policies spanning large time intervals. In contrast, TPM has a high resolution clock. The effective resolution of TPM clock however is reduced when accessed from the CPU. We empirically compared the access latencies to SGX trusted time and TPM secure clock from an application running on CPU. The latency in retrieving SGX time from CSME has a mean of 13msec, whereas TPM clock has an access latency of almost 32msec that is three times more than SGX time latency. Note that both CSME and TPM are separate from the CPU, justifying their large access latencies.

The effective clock resolution that an application sees is either the base timer resolution or the clock access latency, whichever is higher. This makes theoretical time resolution of 1sec for SGX and approximately 32msec for TPM. Although TPM has higher resolution, it cannot be accessed by multiple applications at the same time due to limited TPM resources, thus decreasing it's effective resolution for applications. TPM's clock is mostly used to timestamp stored keys and data inside it, and not designed for timestamping network packets or sensor data. TPM clock does not satisfy the needs of broad applications, hence we choose SGX trusted time as a trusted timer for TIMESEAL's secure time architecture.

`[Challenge 2]` A Secure Path to Trusted Timer: The encrypted SGX time value passes through a secure channel established between PSE and an application enclave. This communication happens via IPC [18]. We know that a compromised OS cannot attack the encrypted and integrity protected packet. We show, however, that it can still cause damage by *delaying* the time packet. Delaying affects timely arrival of of SGX time, hence distorting time for application enclaves. Note that TPM time is also prone to IPC delay attacks.

We implement a delay attack in the OS by delaying all SGX time packets with a random value sampled from a uniform distribution of 0 to 1sec. The result in Figure 1a shows that SGX notion of 1sec fluctuates within 0 to 2.5sec, while 4sec varies between 2.2 to 5.5 sec. Time fuzziness due to delay attacks affects many applications. For example, Timecard [3] servers need to provide consistent response times to clients within few msec of target delays. Fuzzy time in the order of seconds distorts their sense of elapsed time resulting in wastage of compute resources.



(a) Fuzzy SGX time (b) Time dilation & compression

Fig. 1: Two ways to visualize the effect of delay attacks on SGX trusted time

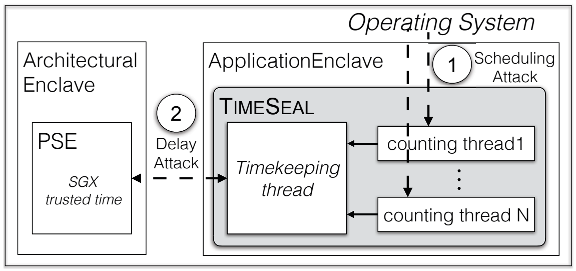


Fig. 2: TIMESEAL components inside an Application Enclave. It is subject to two main attacks: ① OS scheduling attacks on TIMESEAL threads, and ② OS delay attacks on SGX trusted time

Note that a privileged attacker manipulating the Dynamic Voltage and Frequency Scaling (DVFS) achieves the same effect as delay attack. This effect can be visualized in another way shown in Figure 1b. An application relying on SGX time measures 1sec durations on y-axis, which in reality are distorted durations on x-axis. Time advances without a fundamental fixed frequency, resulting in either time dilation or compression across different intervals. Thus, delay or DVFS attacks cause SGX time to dilate and constrict, and we establish that the path to our choice of trusted timer – SGX time – is not secure.

`[Challenge 3]` A Secure Timekeeping Software: A timekeeping service maintained by an enclave process and threads is secure from memory manipulation. However, these threads can still be attacked by malicious OS scheduling that can make time inconsistent. This is because a thread running in enclave mode is the same as a thread running in normal mode from the OS perspective [29], We refer to attacks on timekeeping threads through malicious scheduling as *scheduling attacks*. Note that these attacks also result in time dilation and compression. Hence, we establish that timekeeping software within secure enclave is also prone to attacks.

B. TIMESEAL *Overview*

TIMESEAL is a secure time architecture that overcomes the above mentioned challenges, i.e., the availability of a trusted timer, secure access to that timer, and an attack free timekeeping software. It solves `Challenge 1` by leveraging SGX time derived from a trusted timer. `Challenge 2` is addressed by timely detection and mitigation of delay attacks, while `Challenge 3` is resolved by devising policies that overcome the effect of scheduling attacks.

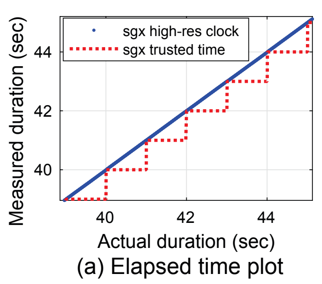
TIMESEAL's components are shown in Figure 2. SGX time provided by PSE to an application enclave is coarse-grained and incapable of detecting and eliminating scheduling and delay attacks. Hence, a high resolution clock is critical to secure time. TIMESEAL provides a high resolution clock comprised of a *timekeeping thread* that keeps track of SGX time, and *counting threads* that interpolate between SGX time to provide sub-sec resolution.

C. Threat Model

The goal of an attacker is to compromise TIMESEAL's sense of time while maintaining stealthiness. Although more prominent attacks could cause damage, e.g., a denial-of-service attack on any component, we consider attacks that have a more enduring impact over time, e.g., an attacker that gradually makes time fuzzy while remaining undetected. Such an attack can successfully establish false proximity for various applications that rely on time for co-location detection [30] [31].

Compromised OS: Our threat model considers TEE by different vendors such as ARM TrustZone and Intel SGX as trustworthy. TIMESEAL does not trust the OS nor hypervisors as they can be corrupted. To maximize damage, an attacker may stay undetected throughout the system's operation because consistent time uncertainty is worse for the system as a single time jump can easily be detected. We also assume that the threads in TIMESEAL are subject to a normal OS scheduling policy and can be aborted/scheduled out at any instant, i.e., they will compete for CPU time with the rest of the system.

Attack vectors: Because the attacker is trying to compromise the timing components of TIMESEAL in a stealthy fashion, the only two attack vectors specific to TIMESEAL's attack surface are the aforementioned (1) *delay attacks* or (2) *scheduling attacks*. For delay attacks, prior knowledge of the system and physical clock characteristics helps the attacker launch an attack that degrades system performance without detection. For example, the



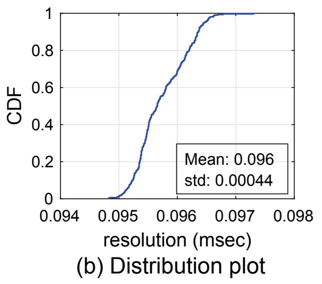


Fig. 3: (a) High-resolution SGX clock measures sub-second durations compared to SGX tick that is only capable of measuring durations greater than a second. (b) Achieved mean resolution of high-res clock is 0.1msec with 0.4microsec standard deviation (std)

OS knows that 'aesmd daemon' encapsulates PSE and handles SGX trusted time packets. Therefore, it launches *delay attacks* on SGX time packets by intercepting all transmitted/received packets to/from aesmd daemon. Hence the attacker is capable of making SGX time fuzzy as established in `Challenge 2`.

It is to be noted that an attack strategy of delaying all SGX packets by a constant value does not harm a system. Adding a constant value to true time does not affect the rate at which time is elapsed. Rather, delaying SGX packets by a different value adds variations to the clock rate and distorts the passage of time. Therefore, our threat model incorporates incremental, random, and distribution based delay attacks on SGX packets.

Detectability of DoS Attacks: The attacker also knows that SGX time increments every sec, referred to as *SGX tick*. It may choose to delay a packet by any arbitrary value. This causes an application polling SGX time to detect missing SGX ticks. We equate this scenario to a denial of service, which becomes an availability issue rather than a security issue. To maintain stealthiness, it would choose to delay by a sec or so. For *scheduling attacks* (established in `Challenge 3`), we assume that an attacker does *not* want to schedule out all threads of a process in order to maintain stealthiness–if there are no threads running, this is considered a detectable denial of service because the count value is less to none.

TimeSeal strives to protect against attackers capable of launching scheduling attacks on SGX enclave threads as well as delay attacks on SGX time packets.

V. High Resolution Secure Clock in SGX

Low resolution of SGX time is not enough for many IoT applications. We provide a *subtick service* that builds a high resolution SGX clock on top of coarse

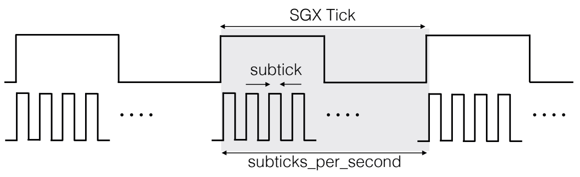


Fig. 4: SGX tick interpolated by subticks

SGX time. We refer to a 1sec SGX time increment as an *SGX tick*. To get fine time granularity, we develop a subtick service that interpolates SGX ticks. This interpolation mechanism uses an SGX tick, which has a large, known one-sec period, and a subtick, which has a short, unknown sub-sec period. SGX tick is used to establish the period of the subtick, and together they provide time with high resolution.

The subtick service is comprised of a *timekeeping thread* inside an enclave that continuously polls SGX time, and a *counting thread* that counts within one SGX tick as shown in Figure 2. Counting thread maintains the number of subticks in one SGX tick, where each subtick is comprised of few instruction cycles as shown in Figure 4. In essence, a subtick is implemented as an internal SGX variable that continually gets incremented every few clock cycles. As this variable is stored inside enclave's protected memory, it cannot be altered or manipulated by a compromised OS. Thus the subtick value is considered trustworthy.

To build a high resolution SGX clock–which we call "high-res clock"–using SGX ticks and subticks, we use a clock model to calculate the current time, i.e., $t_{local} = SGXticks + \frac{subticks}{MA(subticks\_per\_sec)}$, where t_{local} is the local time reported by our high-res clock, $SGXticks$ represents seconds, and $subticks$ divided by the moving mean (MA) of multiple $subticks\_per\_sec$ values in a window represents the fractional part of a sec. Thus, we are able to provide sub-sec clock resolution.

We test the resolution of our high-res clock by measuring fine time intervals. The smallest duration that a clock is able to measure in a stable manner is its resolution. As shown in Figure 3a, the sloped dotted line (blue) shows that the high-res clock is capable of measuring sub-sec time durations. The dashed (red) line shows that SGX time is not capable of measuring durations that are less than a sec, i.e., a new value comes once every sec. Figure 3b shows that our high-res clock is able to achieve a mean resolution of 0.1 msec, i.e., the clock is capable of measuring durations as small as 0.1 msec or timestamp events that are apart by 0.1 msec. This 0.1 msec resolution is a result of software instructions

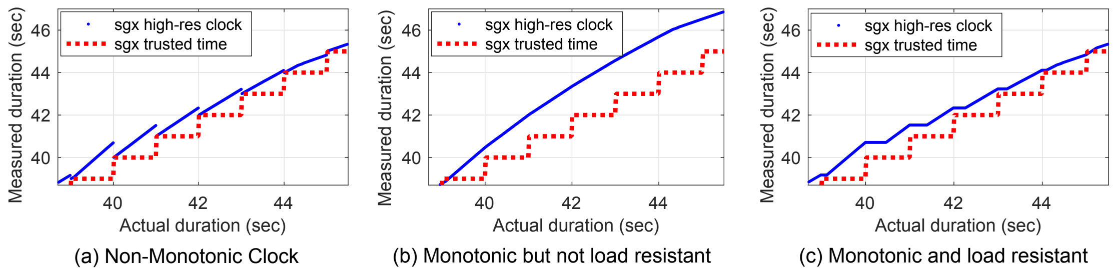


Fig. 5: Comparison of different clock models to build a high resolution SGX clock. A clock should always be monotonic and compensate for time errors during high system load

in subtick service that take up CPU cycles. We can configure the code to improve or relax this resolution.

Threads scheduled under high system load cause subtick variations. We conduct experiments by running a large number of stressing threads along with the subtick service to overload the system by 80%. This causes fluctuations in $subticks\_per\_sec$ that result in discontinuities in t_{local} based on our current clock model. Figure 5a shows a non monotonic clock with time discontinuities shifting the clock back in time. t_{local} exceeds true SGX time as it advances at a rate higher than the nominal rate.

To make the high-res clock monotonic, we revise our clock model by advancing local time from previous local time instead of the latest SGX tick, i.e. $t_{local} = t_{prev\_local} + \frac{subticks}{MA(subticks\_per\_sec)}$. The result of this, as shown in Figure 5b, is a monotonic clock where t_{local} significantly deviates from true time due to accumulated errors over time. We rely on the SGX tick boundary to calculate the accumulated error, $t_{error} = t_{local} - SGXticks$, and compensate for it. *We thus propose a new clock model that not only advances time with respect to previous time, but also takes into account the accumulated error in local time at every SGX tick boundary.*

Adding huge offsets to remove error from time is not a good practice as it can lead to negative durations or high error fluctuations. Thus, we divide this error into smaller chunks equal to the high-res clock resolution. We then remove the error by subtracting small error chunks from local time at every iteration until no error remains. This process of removing error using smaller chunks is called slewing time. Thus, our new clock model is, $t_{local} = t_{prev\_local} + \frac{subticks}{MA(subticks\_per\_sec)} - slew(error)$. Figure 5c shows that the slewed clock is monotonic and slowly converges to true time during peak load.

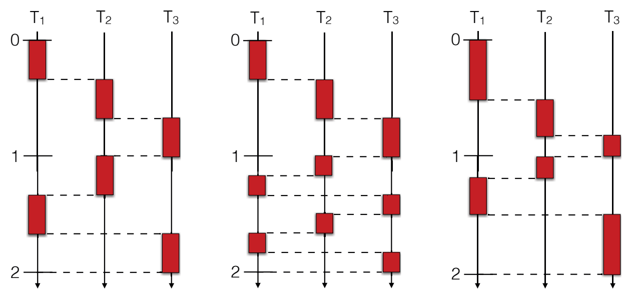
A. Scheduling Attack and Mitigation

An enclave counting thread of the subtick service is continuously counting. This thread is subject to normal OS scheduling policy and can be aborted/scheduled out at any instant. As a result, the number of subticks per sec over multiple SGX ticks are inconsistent. A compromised OS may issue sophisticated attacks and schedule out the counting thread to downgrade the time resolution to seconds, making the subtick service useless for achieving high resolution.

A single counting thread is unlikely to provide a consistent count every sec under malicious scheduling as there is a higher probability that an attacker can identify the counting thread. Our goal is to provide a stable count every sec in the presence of high system load and malicious OS scheduling, both of which may cause huge variations in count values over multiple SGX ticks. We attain this goal by inducing uncertainty in the OS scheduling policy: TimeSeal employs multiple threads and a thread counting policy design that reduces the efficacy of an attack on the time resolution per sec.

1) Thread Counting Policy Design: One naive approach for inducing scheduling uncertainty is to let all threads count all the time and choose one maximum count value at the end of every SGX tick. This is not an effective counting policy because a fair OS scheduling interrupts all threads for the same amount of time, and results in same reduced resolution as with one counting thread. Another approach is to allow only one thread to count at a time for a specific duration before switching to the next thread. If the order in which the threads are scheduled and their count intervals are known, a malicious OS can locate and interrupt the thread that is ready to count. As such, we need to design policies that reduce this predictability.

Policy design variables: The design variables of a thread counting policy include the number of threads, the counting interval assigned to each thread, as well



(a) Policy A (b) Policy B (c) Policy C

Fig. 6: Multiple threads consuming equal CPU time either in one SGX sec or across multiple seconds. Note that threads count in a different order every second

as the order in which each thread counting interval occurs. We first design three policies based on the latter two variables and discuss how varying the number of threads will affect each policy. For the purpose of clarity, Figure 6 depicts the three general approaches for a counting policy consisting of three threads[1]: T_1, T_2, and T_3. Each policy assigns an order and counting interval to each thread. In summary,

- Policy A: assigns a different order but the same count interval to the threads every sec
- Policy B: chooses a different order and count interval every sec
- Policy C: chooses a different thread order every sec while assigning a different count interval to every thread within one sec.

All threads get the same amount of counting time over a small (Policies A and B) or a long period (Policy C). In order to assess the efficacy of each defense, we will first describe possible attack scenarios that aware of TIMESEAL's counting policy design variables.

Attacker strategies: Based on the aforementioned design variables of TIMESEAL, a clever attacker may craft one of the three following approaches:

- Attack 1: Choose n out of N counting threads randomly to be scheduled out for one sec, where n could be any value from 1 to $N-1$.
- Attack 2: Schedule out all counting threads for the same interval delay of c_d secs one after the other in any order, where $c_d < 1$.
- Attack 3: Schedule out all threads for the same interval delay of c_d secs at the same time.

In all cases, we assume the attacker not only knows the design variables of TIMESEAL, but can also identify the candidate set of counting threads of the associated application. In reality, there may be several other threads associated with the application that may further obfuscate the counting process. Note that an attacker scheduling out all threads yields a zero count value and is indicative of an attack. Thus an attack of this nature is considered a detectable denial of service and out of the scope for this analysis.

[1]Note that this figure only depicts the counting threads for clarity. These threads will most likely compete with the rest of the OS.

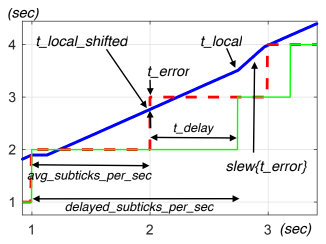
Policy efficacy: To assess the efficacy of each policy against each defense, we define a degradation metric, D, as the portion of $subticks$ that will be omitted from the overall $subticks$ count across one SGX tick, e.g., if $subticks\_per\_tick = 1000$, and $D= 0.5$, that would mean an attack caused the subtick service to lose 50% of its resolution. We further determine what is the maximum and minimum degradation an attack may achieve, D_{max} and D_{min}, respectively, as well as the probability of achieving the maximum degradation, P(D).

Table I summarizes the results of our formalization for each attack's efficacy across all policies. In general, Attack 3 is the most powerful and has complete control of the subtick count degradation, D, but causes a detectable denial of service. Attack 1 has the most consistent degradation under stealthy conditions assuming that an attacker can accurately identify the set of counting threads. Attack 2 has the highest possible degradation but with a much lower probability of success.

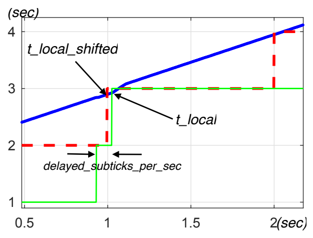
All scheduling policies in TimeSeal randomize thread order but differ in choosing their count intervals per second. In terms of choosing a policy, the performance of Policy B is comparable to Policy A across all attacks except that Policy B also decreases the attack's success probability. Policy C performs worse because of uneven count distribution among threads per second, and it is possible that an attacker schedules out the threads with the largest count intervals, hence increasing the probability of resolution degradation and causing more damage. As such, we hypothesize that Policy B will be the most robust approach against scheduling attacks. We will validate this hypothesis empirically in Section VI.

	Policy A			**Policy B**			**Policy C**		
	P(D)	D_{max}	D_{min}	**P**(D)	D_{max}	D_{min}	**P**(D)	D_{max}	D_{min}
A1	1	$\frac{n}{N}$	$\frac{n}{N}$	1	$\frac{n}{N}$	$\frac{n}{N}$	1	$\frac{n}{N}$	$\frac{n}{N}$
A2	$(\frac{1}{N})^N$	1	0	$(\frac{1}{N})^N$	1	0	$(\frac{1}{N})^N$	<1	0
A3	1	c_d	c_d	1	c_d	c_d	1	c_d	c_d

TABLE I: Efficacy of each attack against all policy designs enumerated in Figure 6. The degradation metric D refers to the portion of subticks that will be lost due to the attack over one SGX tick



(a) Condition 1: delayed tick duration is much greater than true tick duration



(b) Condition 2: delayed tick duration is much smaller than true tick duration

Fig. 7: Measured local time (blue dotted line) is slewed by restoring true SGX ticks (dashed red line) from delayed ticks (solid green line)

Algorithm 1 Delay Free Clock Model

```
procedure RestoreTrueSGXTicks(avg_subticks_per_sec, subticks_persec_upperlimit, subticks_persec_lowerlimit)
    $a \leftarrow$ avg_subticks_per_sec
    $u \leftarrow$ subticks_persec_upperlimit
    $l \leftarrow$ subticks_persec_lowerlimit
    $true\_tick\_dist \leftarrow 1$
new delayed tick:
    $d \leftarrow delayed\_subticks\_per\_sec$
    if $d > u$ then                    ▷ true tick found: Condition 1
        $true\_tick\_dist \leftarrow 1$
    else if $d < u$ & $d > l$ then              ▷ true tick passed
        $true\_tick\_dist \leftarrow true\_tick\_dist + 1$
    else if $d < l$ then               ▷ true tick found: Condition 2
        $true\_tick\_dist \leftarrow 1$
        $d \leftarrow a$
    end if
    $n \leftarrow true\_tick\_dist$
    $t_{delay} \leftarrow \frac{\sum_1^n d}{a} - n$
    $t_{local\_shifted} \leftarrow t_{local} - t_{delay}$
    $t_{error} \leftarrow t_{local\_shifted} - SGXticks$
    goto new delayed tick
end procedure
procedure Advance Time($t_{prev\_local}$, avg_count)
    $t_{error} \leftarrow procedure$\{Restore True SGX TICKS\}
new subtick:
    $t_{local} \leftarrow t_{prev\_local} + \frac{subticks}{avg\_count} - slew\{t_{error}\}$
    goto new subtick
end procedure
```

B. Overcoming SGX Delay Attacks

Referring back to TimeSeal's design overview in Figure 2, we provide a high resolution clock by overcoming SGX trusted time limitations. Assuming we choose an optimal scheduling policy–tentatively, Policy B–that provides stable subtick values across stealthy attacks, we can also use it to detect delay attacks. If an attacker delays one SGX time packet by n^{th} of a sec, the previous SGX tick is $1 + n$ sec away from the current tick, while the next one will be $1 - n$ sec away from the current tick. A stealthy attacker that wants to avoid detection delays SGX time packets by a little bit more than a sec and makes SGX time fuzzy by twice the delay as established in Figure 1a. The effect of a delay attack is time dilation and compression. Some elapsed seconds span more than a few seconds, while other seconds only span a few msecs as shown in Figure 1b. As a result, a time-aware application that relies on the physical notion of elapsed time calculates wrong intervals [31].

Our design relies on the intuition that subticks over multiple seconds show large variations under delay attacks. The attacker cannot avoid these large variations even when it launches coordinated delay and scheduling attacks. With no knowledge of the time value inside an SGX packet, the attacker does not know when a new SGX tick starts. There are two ways an attacker can avoid subtick variations: it can delay packets by a small value–which reduces the time error, or it can delay all packets by the same value–which is not an attack because relative time stays the same. Both ways do not result in an attack.

To provide an accurate high resolution clock, the local time t_{local} should join the SGX ticks by a straight line. Figure 7a, 10a shows that t_{local} significantly deviates from true SGX ticks due to delay attacks. The first step to overcome delay attacks and align t_{local} with true SGX ticks is to approximately locate these ticks. We make an observation that large delay variation gives rise to two conditions that are an indication of a delayed tick being close enough to true SGX tick. `Condition 1`, as shown in Figure 7, arises when $delayed\_subticks\_per\_sec$ is large enough for a delayed tick to be aligned with at least one of the true SGX ticks. Once a true tick is identified, we shift the delayed tick back to true tick by t_{delay}. This delay approximately equals $delayed\_subticks\_per\_sec$ minus the average of delayed subticks $avg\_subtick\_per\_sec$. We can find the error in local time t_{error} by subtracting t_{delay} from t_{local} and shifting it to true SGX Tick at $t_{local\_shifted}$. This error helps slew the clock to true time. Figure 7b shows a scenario that gives rise to `Condition 2` of detecting a true SGX tick., This condition states if $delayed\_subticks\_per\_sec$ is small enough, the delayed tick overlaps the true tick within an error tolerance. In that case t_{delay} is zero and t_{error} would be the difference in t_{local} and true SGX tick. Algorithm 1 provides the details of how we calculate t_{delay} and t_{error} to overcome delay attacks and slew the clock towards true time.

Calculating the right t_{delay} value is critical to construct a delay-free clock. Algorithm 1 shows in detail how t_{delay} is calculated from different set of parameters.

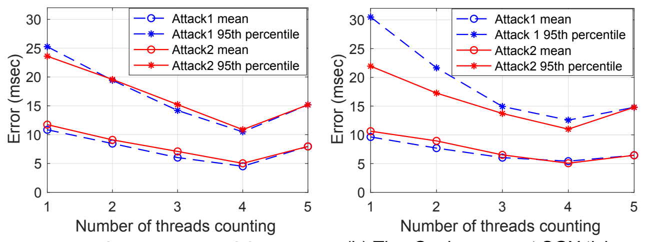
The range of delay variation helps select values for these parameters: $subticks\_persec\_upperlimit$ determines how large $delayed\_subticks\_per\_sec$ should be to satisfy `Condition 1`, $subticks\_persec\_lowerlimit$ satisfies `Condition 2`, whereas $avg\_subticks\_per\_sec$ is the mean of multiple $delayed\_subticks\_per\_sec$. This mean provides a good approximation of true $subticks\_per\_sec$. The upper and lower limits parameters are adjusted based on the maximum, minimum, and average $delayed\_subticks\_per\_sec$.

To align t_{local} with true SGX ticks, it is necessary to find a true tick and extrapolate time from there. As the occurrence of Condition 1 and 2 is not high, we maintain a parameter $true\_tick\_dist$ that determines how many delayed ticks have elapsed since the last found true tick. In essence, it makes sure that t_{delay} is always calculated with respect to true tick. In reality, error in time gets accumulated due to inaccuracies in t_{delay} calculation with every passing delayed tick since the true tick. Therefore, the $true\_tick\_dist$ value should not exceed an accumulated error threshold that deems the time unreliable.

VI. Implementation and Evaluation

We provide a scalable TimeSeal implementation on a SGX enabled computer with an i7-6700K processor and 16 GiB memory running Ubuntu Linux 16.04.3, kernel version 4.10.32. The $sgx\_get\_trusted\_time$ API provides time from the hardware management engine. An application that wishes to acquire secure time instantiates TimeSeal within its own process to limit OS interactions and avoid delay attacks. Counting policies and their associated parameters, e.g., the number of threads, counting order, and count values, are maintained within the SGX process memory. To randomize the selection of threads, and count values every SGX sec, we use sgx_read_rand API to generate a true random number. Every thread increments its own counter based on the counting policy instead of incrementing a common global counter. This is to avoid reliance on OS mutual exclusion locks for race conditions. To manipulate thread count, OS may give two threads the lock at once or may not give a lock to any thread at all [32].

Evaluation metrics: We choose two evaluation metrics. One metric is the time difference of TimeSeal's high resolution secure clock with SGX trusted time at the boundary of a true SGX tick. We term this accumulated error per SGX tick as $error_{tick}$. It represents frequency error in TimeSeal's clock. Note that TimeSeal's clock is derived from SGX time by constantly polling it. The SGX time access latency is around 13 msec



(a) TimeSeal error w.r.t OS time (b) TimeSeal error w.r.t SGX ticks

Fig. 8: Scheduling attacks 1 and 2 aborts either 1, 2, 3, or 4 threads out of 5 counting threads. Policy B bounds the error to within tens of msec

on average. Therefore, $error_{tick}$ would always have a standard deviation comparable to access latency.

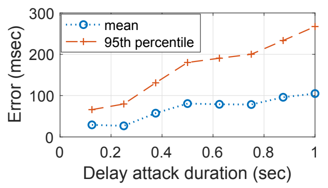
The other evaluation metric compares TimeSeal's jitter with respect to OS time. We term it as $error_{OS}$. Considering an OS's monotonic clock as the ground truth for evaluation purposes, we generate small durations with respect to OS time. TimeSeal timestamps these durations and the resultant jitter indicates its stability w.r.t. OS's monotonic clock. Because the oscillators for OS's clock and TimeSeal's clock are different, there will be sub millisec level relative drift between both clocks. However, this drift is masked by millisec level access latencies.

Countering scheduling attacks: Table II provides a summary of TimeSeal's errors in the presence of scheduling attacks. We run different experiments with three threads (N = 3) counting under different policies (A, B, C) experiencing the three categories of scheduling attacks (1, 2, 3) under 50% system load. The performance of Policy B is comparable to Policy A in terms of empirical errors except that Policy B also decreases the attack's success probability. Policy C performs worse when few threads are allowed to run because of uneven count distribution among threads per sec. Lastly, Attack 3 degrades all policies equally because it causes a detectable denial of service for a duration it is launched in. Figure 8 shows the relationship between the number of threads and TimeSeal's errors. Using Policy B, we see a decrease in errors with an increase in number of threads because of small attack success probability. Also note that 95th percentile $error_{tick}$ for Attack2 is smaller as compared to Attack 1 because of its lower attack success probability as discussed in Section V-A.

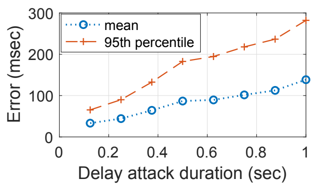
Countering delay attacks: Time error is directly proportional to delay attack duration. The more the SGX time packet is delayed, the more error an attacker can accumulate. We test different delay attack intervals rang-

	Attack 1				**Attack 2**				**Attack 3**	
	Single Thread		N-1 Threads		Single Thread		N-1 Threads		50% c_d	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Policy A	0.5|| 3	7|| 7	0.5|| 3	9|| 8.4	0.5|| 1.8	7|| 7	0.5|| 3	9|| 8.5	0.6|| 8	25|| 24
Policy B	0.002|| 3	6.6|| 5.2	0.068|| 3	9.6|| 8.5	0.076|| 1.5	6|| 5	0.05|| 0.6	10|| 7.4	0.08|| 9.4	22|| 24
Policy C	8|| 2.7	300|| 200	0.3|| 0.6	1500|| 1600	1|| 11	81|| 60	1|| 25	100|| 99	0.02|| 5	31|| 26

TABLE II: Counting policies results for single and multiple-thread contexts, where μ and σ are the mean and standard deviation of induced error in milliseconds. A "Single" Thread attack implies only one thread is scheduled out while an "(N-1) Threads" attack implies only one thread is counting at a time. Format of error is $\{error_{tick} \parallel error_{OS}\}$



(a) TimeSeal error w.r.t OS time



(b) TimeSeal error w.r.t SGX ticks

Fig. 9: Delay attacks of different durations

ing from 0 to 1sec under 50% system load. In Figure 9, although TIMESEAL's error increases with an increase in delay duration, our delay mitigation technique bounds it to be within 100s of msec. For example, for a delay duration of 1sec, $error_{tick}$ has a 140msec mean and 342msec 95th percentile. $error_{OS}$ has 137msec mean with 356msec 95th percentile. We can also detect and bound delay attacks above 1sec by adjusting the subtick related parameters discussed in Section V-B.

Overcoming scheduling and delay attacks: Figure 10 shows the effects of delay attacks and scheduling attacks on time plots. SGX true time advances every sec (red dashed line), delayed SGX time advances with variations around 1 sec (green solid line), and TIMESEAL's clock advances with a msec resolution (blue dotted line). Delay attacks distort frequency of TIMESEAL's clock such that it advances at a different rate every sec as shown in Figure 10a.

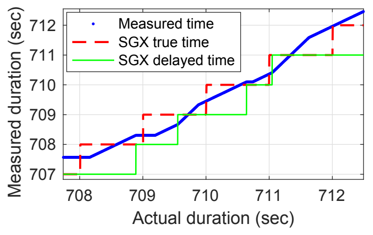
Our delay mitigation technique restores true SGX ticks, adjusts TIMESEAL's frequency, and slews accumulated errors of delay attacks. By doing so, Figure 10b shows that the high resolution TIMESEAL clock traces SGX ticks precisely and advances with a stable frequency. If an attacker also launches scheduling attacks on top of delaying SGX packets, our policies make sure that the error remains bounded to within 100s of msecs. Depending upon the scheduling attack type and c_d value per thread, Figure 10c's zoomed-in plot shows a decrease in TIMESEAL's effective resolution. The wavy plot with small time discontinuities is a result of different threads counting at different times due to scheduling attacks. TIMESEAL's resolution degradation is much less than the clock errors. Figure 11 presents the mean $error_{tick}$ and $error_{OS}$ distributions for different scheduling attack types and 1sec delay attacks. The mean of errors are a result of delay attacks while the interquartiles (iqr) are a result of scheduling attack. Note that fewer number of threads yield slightly large errors. For Attack 1 with N-1 threads counting ($A1 : (N-1)c$), the mean $error_{os}$ is 135msec with 11msec iqr, and mean $error_{tick}$ is 138msec with 9msec iqr. For Attack 1 with only one thread counting ($A1 : c$) the mean errors increase to 165msec with 15msec iqr for both errors.

System resources overhead: TIMESEAL threads are not given a high priority and they are scheduled as normal threads. Systems under high load may give less CPU time to TIMESEAL threads resulting in errors due to varying $subticks\_per\_sec$ as shown in Figure 5. Scheduling attack achieves same degradation maliciously. Hence, we argue that our clock model and counting policies are equally resilient to high load scenarios and an attacker can't obfuscate an attack during high load.

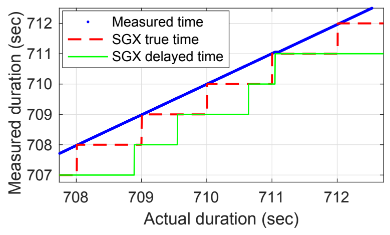
For an application that needs secure time, TIMESEAL's model of polling PSE for SGX time is similar to current SGX model. To enforce certain time based policies, SGX enclaves also poll PSE continuously to make sure that a certain duration has passed [33]. Therefore, we argue that there is no bandwidth increase over current SGX use cases. However, to decrease the probability of an attack's success, more counting threads under any scheduling policy are employed, thus increasing the CPU usage. We argue that securing time is a big challenge and our design exercise highlights the tradeoff among performance, resource consumption, and security.

VII. DISCUSSION

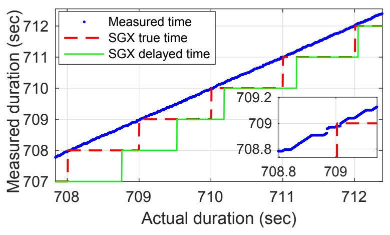
TIMESEAL placement: TIMESEAL's components are enclosed inside application enclaves to avoid delay attacks on additional communication channels via the OS. This design requires applications to have their own instances of TIMESEAL threads. Another possibility to avoid delay attacks between PSE and TIMESEAL is to



(a) Delay attack

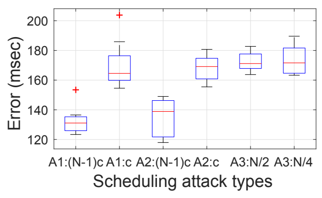


(b) Delay attacks compensation

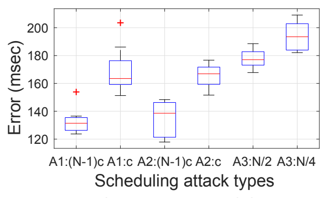


(c) Both attacks compensation

Fig. 10: Time plot of true SGX, delayed SGX, and TIMESEAL measured time. (a) Error in $subticks\_per\_sec$ due to delayed SGX ticks affects TIMESEAL's accuracy. (b) Delay attack mitigation technique, connects measured time to true SGX ticks. (c) Scheduling attacks on top of delay attacks decrease resolution as shown in zoomed-in plot



(a) TimeSeal error w.r.t OS time



(b) TimeSeal error w.r.t SGX ticks

Fig. 11: An attacker delays SGX packets by 1 sec and launches scheduling attacks 1, 2, 3. The mean error distribution for all attacks remain within 100s of msec

provide a high resolution clock within PSE. A modified PSE, though, will not be able to use a secure channel with CSME because the root of trust isn't established by Intel servers [34].

TIMESEAL for other TEE: TIMESEAL's architecture and design principles are not dependent on one TEE. Though we used SGX, any TEE that provides access to a trusted timer can be considered. Exploring the possibility of enabling TIMESEAL for ARM TrustZone is in consideration as it dominates the embedded market.

Recommendations for vendors: To avoid complex and incomplete designs of secure clocks–which is still an open problem–we list a set of requirements for hardware vendors that fulfill all conditions of a secure clock to the best of our knowledge. First, there should exist a hardware timer with associated registers that no privileged hardware or software can write to. Second, this timer should be derived from a high frequency oscillator that should not be overclocked or under-clocked by a malicious software. Third, the timer should have a sufficient number of bits–preferably 64 bits–so that it never overflows. Finally, access to the timer value should be securely memory mapped for fast access and independent of OS manipulation and delays. These requirements can only be fulfilled by a hardware vendor.

VIII. CONCLUSION

Securing time in an untrusted OS is a challenge. We present TIMESEAL, a new architecture that leverages TEE for hardware timer protection and eliminates timing limitations and vulnerabilities in TEE to secure time. TIMESEAL provides a secure local clock that is good for measuring durations. There are a plethora of applications that require secure global time [21] [35] [36]. Researchers have addressed global clock security by protecting time transfer packets in the network. This is an orthogonal area of research, and we plan to synchronize TIMESEAL to global reference.

TIMESEAL protects time or in other words, "seal" time so that no privileged adversary can arbitrarily change the notion of time, and compromise the safety and performance of applications.

ACKNOWLEDGEMENT

The research reported in this paper was sponsored in part by the National Science Foundation (NSF) under award # CNS-1329755 and CNS-1705135, by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by the Army Research Laboratory (ARL) under Cooperative Agreement W911NF-17-2-0196. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ARL, DARPA, NSF, SRC, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] C. Meadows, R. Poovendran, D. Pavlovic, L. Chang, and P. Syverson, "Distance bounding protocols: Authentication logic analysis and collusion attacks," in *Secure localization and time synchronization for wireless sensor and ad hoc networks*. Springer, 2007, pp. 279–298.

[2] R. Tachet, P. Santi, S. Sobolevsky, L. I. Reyes-Castro, E. Frazzoli, D. Helbing, and C. Ratti, "Revisiting street intersections using slot-based systems," *PloS one*, vol. 11, no. 3, p. e0149607, 2016.

[3] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan, "Timecard: Controlling user-perceived delays in server-based mobile applications," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 85–100.

[4] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM, 2013, p. 20.

[5] C. Lee, C. Park, K. Jang, S. Moon, and D. Han, "Accurate latency-based congestion feedback for datacenters," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, 2015, pp. 403–415. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/lee-changhyun

[6] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with déjá vu," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 7–18.

[7] S. Hamilton, D. Sengupta, and R. Gupta, "Introducing automatic time stamping (ats) with a reference implementation in swift," in *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2018, pp. 138–141.

[8] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, p. 8, 2015.

[9] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. Stillwell *et al.*, "Scone: Secure linux containers with intel sgx." in *OSDI*, vol. 16, 2016, pp. 689–703.

[10] S. Shinde, D. Tien, S. Tople, and P. Saxena, "Panoply: Low-tcb linux applications with sgx enclaves," in *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2017, p. 12.

[11] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library os for unmodified applications on sgx," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017, p. 8.

[12] J. Seibel, K. LaFlamme, F. Koschara, R. Schumak, and J. Debate, "Trusted execution environment," Jul. 27 2017, uS Patent App. 15/007,547.

[13] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon *et al.*, "ftpm: A software-only implementation of a tpm chip." in *USENIX Security Symposium*, 2016, pp. 841–856.

[14] H. Liang, M. Li, Q. Zhang, Y. Yu, L. Jiang, and Y. Chen, "Aurora: Providing trusted system services for enclaves on an untrusted system," *arXiv preprint arXiv:1802.03530*, 2018.

[15] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer, "Shieldbox: Secure middleboxes using shielded execution," in *Proceedings of the Symposium on SDN Research*. ACM, 2018, p. 2.

[16] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 3–24.

[17] linuxsgx, "Intel(r) software guard extensions for linux os, trusted time discussion," https://github.com/intel/linux-sgx/issues/161, 2018.

[18] S. Cen and B. Zhang, "Trusted time and monotonic counters with intel sgx platform services," https://software.intel.com/sites/default/files/managed/1b/a2/Intel-SGX-Platform-Services.pdf, 2017.

[19] B. Trach, A. Krohmer, S. Arnautov, F. Gregor, P. Bhatotia, and C. Fetzer, "Slick: Secure middleboxes using shielded execution," *arXiv preprint arXiv:1709.04226*, 2017.

[20] S. M. Kim, J. Han, J. Ha, T. Kim, and D. Han, "Enhancing security and privacy of tor's ecosystem by using trusted execution environments." in *NSDI*, 2017, pp. 145–161.

[21] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *Proceedings of the 2016 aCM sIGSAC conference on computer and communications security*. ACM, 2016, pp. 270–282.

[22] R. Kotla, T. Rodeheffer, I. Roy, P. Stuedi, and B. Wester, "Pasture: Secure offline data access using commodity trusted hardware." in *OSDI*, 2012, pp. 321–334.

[23] C. Chen, H. Raj, S. Saroiu, and A. Wolman, "ctpm: A cloud tpm for cross-device trusted applications." in *NSDI*, 2014, pp. 187–201.

[24] H. Liang and M. Li, "Bring the missing jigsaw back: Trusted-clock for sgx enclaves," in *Proceedings of the 11th European Workshop on Systems Security*. ACM, 2018, p. 8.

[25] D. Kohlbrenner and H. Shacham, "Trusted browsers for uncertain times." in *USENIX Security Symposium*, 2016, pp. 463–480.

[26] M. Herlihy and J. E. B. Moss, *Transactional memory: Architectural support for lock-free data structures*. ACM, 1993, vol. 21.

[27] intel sgx, "Intel(r) software guard extensions software developer manual," https://www.intel.com/content/dam/www/public/us/en/documents/manuals, 2016.

[28] intel sgx forums, "Intel(r) software guard extensions forum, tsc discussion topic 743186," https://software.intel.com/, 2017.

[29] Intel, "Intel(r) developer zone, enclave thread scheduling discussion, topic 635939," https://software.intel.com/en-us/forums/, 2018.

[30] J. Han, A. J. Chung, M. K. Sinha, M. Harishankar, S. Pan, H. Y. Noh, P. Zhang, and P. Tague, "Do you feel what i hear? enabling autonomous iot device pairing using different sensor types," in *Do You Feel What I Hear? Enabling Autonomous IoT Device Pairing using Different Sensor Types*. IEEE, 2018, p. 0.

[31] D. Singelee and B. Preneel, "Location verification using secure distance bounding protocols," in *Mobile Adhoc and Sensor Systems Conference, 2005. IEEE International Conference on*. IEEE, 2005, pp. 7–pp.

[32] D. R. Ports and T. Garfinkel, "Towards application security on untrusted operating systems." in *HotSec*, 2008.

[33] linux sgx, "Intel(r) software guard extensions for linux os, drm sample code," https://github.com/intel/linux-sgx/tree/master/SampleCode/SealedData, 2018.

[34] linuxsgx, "Intel(r) software guard extensions for linux os, pse modification discussion," https://github.com/intel/linux-sgx/issues/194, 2018.

[35] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

[36] S. D'souza and R. Rajkumar, "Time-based coordination in geo-distributed cyber-physical systems," in *Proceedings of the 9th USENIX Conference on Hot Topics in Cloud Computing*. USENIX Association, 2017, pp. 9–9.